architecture

Overview Separation of Concerns ■ Container Components : Handle business logic and data management Presentational Components : Focus purely on rendering and user interaction Services : Manage data operations and external API communication Unidirectional Data Flow ■ Services → Container Components → Presentational Components → User Actions → Container Components → Services Modern Angular Patterns ■ Angular Signals for reactive state management Standalone Components for better tree-shaking OnPush Change Detection for optimal performance TypeScript-first approach with strong typing Component Types ■ 1. Container Components ■ Purpose : Handle business logic, data management, and state coordination. Characteristics : Inject and use services Manage application state with signals Handle loading, error, and data states Coordinate between services and presentational components Transform data for presentation Example Structure : @ Component ({ ... }) export class UserListContainerComponent { private userService = inject (UserService) ; protected users = signal < User [ ] > ([ ]) ; protected loading = signal (false) ; protected error = signal < Error | null > (null) ; protected userData = computed (( ) => this . users ( ) . map (user => ({ id : user . id , title : user . name , description : user . email , isActive : user . isActive }))) ; constructor ( ) { this . loadUsers ( ) ; } private loadUsers ( ) : void { this . loading . set (true) ; this . userService . getData ( ) . pipe (takeUntilDestroyed ( )) . subscribe ({ next : (users) => { this . users . set (users) ; this . loading . set (false) ; } , error : (error) => { this . error . set (error) ; this . loading . set (false) ; } }) ; } } 2. Presentational Components ■ Purpose : Focus purely on rendering and user interaction. Characteristics : Receive data through inputs Emit events through outputs No direct service dependencies Highly reusable and testable Built-in accessibility features Example Structure : @ Component ({ ... }) export class UserCardComponent { readonly data = input . required < UserCardData > ( ) ; readonly disabled = input < boolean > (false) ; readonly actionTriggered = output < string > ( ) ; readonly itemClicked = output < UserCardData > ( ) ; readonly isInteractive = computed (( ) => ! this . disabled ( ) && this . data ( ) . isActive) ; protected onClick ( ) : void { if (this . isInteractive ( )) { this . itemClicked . emit (this . data ( )) ; } } } 3. Services ■ Purpose : Handle data operations and external API communication. Characteristics : CRUD operations for data management Reactive state with BehaviorSubject Comprehensive error handling Caching strategies Type-safe interfaces Example Structure : @ Injectable ({ providedIn : "root" }) export class UserService { private apiUrl = "/api/users" ; private dataSubject = new BehaviorSubject < User [ ] > ([ ]) ; public data$ = this . dataSubject . asObservable ( ) ; getData ( ) : Observable < User [ ] > { return this . http . get < User [ ] > (this . apiUrl) . pipe (map ((data) => { this . dataSubject . next (data) ; return data ; }) , catchError (this . handleError)) ; } create (user : Omit < User , "id" >) : Observable < User > { return this . http . post < User > (this . apiUrl , user) . pipe (map ((newUser) => { const currentData = this . dataSubject . value ; this . dataSubject . next ([... currentData , newUser]) ; return newUser ; }) , catchError (this . handleError)) ; } } Data Flow Patterns ■ 1. Data Loading Flow ■ // 1. Container component loads data export class UserListContainerComponent { private userService = inject (UserService) ; protected users = signal < User [ ] > ([ ]) ; constructor ( ) { this . loadUsers ( ) ; } private loadUsers ( ) : void { this . userService . getData ( ) . pipe (takeUntilDestroyed ( )) . subscribe ({ next : (users) => this . users . set (users) , error : (error) => console . error ("Failed to load users:" , error) , }) ; } } 2. Data Transformation Flow ■ // 2. Container transforms data for presentation export class UserListContainerComponent { protected userData = computed (( ) => this . users ( ) . map ((user) => ({ id : user . id , title : user . name , description : user . email , isActive : user . isActive , }))) ; } 3. User Interaction Flow ■ // 3. Presentational component emits events export class UserCardComponent { readonly itemClicked = output < UserCardData > ( ) ; protected onClick ( ) : void { this . itemClicked . emit (this . data ( )) ; } } // 4. Container handles events export class UserListContainerComponent { protected onUserSelect (userData : UserCardData) : void { // Handle user selection this . router . navigate (["/users" , userData . id]) ; } } State Management Strategy ■ Signal-Based State ■ // Container component state export class ExampleContainerComponent { // Core state protected data = signal < Data [ ] > ([ ]) ; protected loading =

signal (false) ; protected error = signal < Error | null > (null) ; // Computed state protected hasData = computed (( ) => this . data ( ) . length > 0) ; protected canLoadMore = computed (( ) => ! this . loading ( ) && ! this . error ( ) && this . hasData ( )) ; // Transformed state for presentation protected presentationData = computed (( ) => this . data ( ) . map ((item) => this . transformForPresentation (item))) ; } Reactive Updates ■ // Services provide reactive streams export class ExampleService { private dataSubject = new BehaviorSubject < Data [ ] > ([ ]) ; public data$ = this . dataSubject . asObservable ( ) ; getData ( ) : Observable < Data [ ] > { return this . http . get < Data [ ] > (this . apiUrl) . pipe (map ((data) => { this . dataSubject . next (data) ; // Update cache return data ; })) ; } } Component Communication Patterns ■ 1. Parent-Child Communication ■ // Container passes data to presentational < app - user - card [data] = "userData()" [disabled] = "loading()" (actionTriggered) = "onUserAction($event)" (itemClicked) = "onUserSelect($event)" > < / app - user - card > 2. Service-Based Communication ■ // Multiple containers share data through service export class UserListContainerComponent { private userService = inject (UserService) ; protected users = signal < User [ ] > ([ ]) ; constructor ( ) { this . userService . data$. pipe (takeUntilDestroyed ( )) . subscribe ((users) => this . users . set (users)) ; } } export class UserDetailContainerComponent { private userService = inject (UserService) ; protected selectedUser = signal < User | null > (null) ; constructor ( ) { this . userService . data$. pipe (takeUntilDestroyed ( )) . subscribe ((users) => { // Find selected user const user = users . find ((u) => u . id === this . route . snapshot . params ["id"]) ; this . selectedUser . set (user || null) ; }) ; } } Error Handling Strategy ■ 1. Service-Level Error Handling ■ export class ExampleService { getData ( ) : Observable < Data [ ] > { return this . http . get < Data [ ] > (this . apiUrl) . pipe (catchError ((error) => { console . error ("Service error:" , error) ; throw error ; })) ; } } 2. Container-Level Error Handling ■ export class ExampleContainerComponent { protected error = signal < Error | null > (null) ; private loadData ( ) : void { this . loading . set (true) ; this . error . set (null) ; this . exampleService . getData ( ) . pipe (takeUntilDestroyed ( )) . subscribe ({ next : (data) => { this . data . set (data) ; this . loading . set (false) ; } , error : (error) => { this . error . set (error) ; this . loading . set (false) ; } , }) ; } protected retry ( ) : void { this . loadData ( ) ; } } 3. User-Friendly Error Display ■ @if (error(); as errorData) { < div class = " error-state " role = " alert " > < h2 > Something went wrong </ h2 > < p > {{ errorData.message }} </ p > < button (click) = " retry() " > Try Again </ button > </ div > } Performance Optimization ■ 1. OnPush Change Detection ■ @ Component ({ changeDetection : ChangeDetectionStrategy . OnPush , }) export class ExampleComponent { // Component only updates when inputs change } 2. Signal-Based Reactivity ■ // Only updates when dependencies change protected computedValue = computed (( ) => this . data ( ) . filter (item => item . isActive) . length) ; 3. Lazy Loading ■ // Load data only when needed export class LazyContainerComponent { private loaded = false ; protected loadData ( ) : void { if (! this . loaded) { this . loaded = true ; this . exampleService . getData ( ) . subscribe ( ) ; } } } Testing Strategy ■ 1. Presentational Component Testing ■ describe ("UserCardComponent" , ( ) => { let component : UserCardComponent ; beforeEach (( ) => { component = new UserCardComponent ( ) ; }) ; it ("should emit click event when interactive" , ( ) => { const spy = jasmine . createSpy ( ) ; component . itemClicked . subscribe (spy) ; component . data . set ({ id : "1" , title : "Test" , isActive : true }) ; component . disabled . set (false) ; component . onClick ( ) ; expect (spy) . toHaveBeenCalled ( ) ; }) ; }) ; 2. Container Component Testing ■ describe ("UserListContainerComponent" , ( ) => { let component : UserListContainerComponent ; let userService : jasmine . SpyObj < UserService > ; beforeEach (( ) => { const spy = jasmine . createSpyObj ("UserService" , ["getData"]) ; TestBed . configureTestingModule ({ providers : [{ provide : UserService , useValue : spy }] , }) ; component = TestBed . createComponent (UserListContainerComponent) . componentInstance ; userService = TestBed . inject (UserService) ; }) ; it ("should load users on init" , ( ) => { const mockUsers = [{ id : "1" , name : "John" }] ; userService . getData . and . returnValue (of (mockUsers)) ; component . ngOnInit ( ) ; expect (component . users ( )) . toEqual (mockUsers) ; }) ; }) ; 3. Service Testing ■ describe ("UserService" , ( ) => { let service : UserService ; let httpMock : HttpTestingController ; beforeEach (( ) => { TestBed . configureTestingModule ({ imports : [HttpClientTestingModule] , providers : [UserService] , }) ; service = TestBed . inject (UserService) ; httpMock = TestBed . inject (HttpTestingController) ; }) ; it (

"should fetch users" , ( ) => { const mockUsers = [{ id : "1" , name : "John" }] ; service . getData ( ) . subscribe ((users) => { expect (users) . toEqual (mockUsers) ; }) ; const req = httpMock . expectOne ("/api/users") ; req . flush (mockUsers) ; }) ; }) ; Best Practices ■ 1. Component Design ■ Single Responsibility : Each component has one clear purpose Composition over Inheritance : Use composition for reusability Props Down, Events Up : Data flows down, events flow up Pure Functions : Presentational components should be pure 2. State Management ■ Single Source of Truth : Services are the source of truth Immutable Updates : Use signals for immutable state updates Computed Values : Use computed for derived state Error Boundaries : Handle errors at appropriate levels 3. Performance ■ OnPush Change Detection : Use for all components Signal-Based Reactivity : Leverage Angular signals Lazy Loading : Load data only when needed Memory Management : Use takeUntilDestroyed for cleanup 4. Testing ■ Unit Tests : Test components in isolation Integration Tests : Test component interactions Service Tests : Mock HTTP requests Accessibility Tests : Test with screen readers Migration Guide ■ From Traditional Angular ■ Replace Services : Update to use BehaviorSubject pattern Add Signals : Replace properties with signals Update Components : Split into container/presentational Add Error Handling : Implement comprehensive error handling Update Tests : Write tests for new patterns From Other Patterns ■ Redux to Signals : Replace Redux with signal-based state NgRx to Services : Use services instead of NgRx Smart/Dumb Pattern : Align with container/presentational pattern Developer Checklist ■ Before Implementing Architecture: ■ Are presentational components pure (no business logic, only rendering)? Do container components handle all data management and business logic? Are all state updates using Angular Signals? Is OnPush change detection enabled on all components? Do services use BehaviorSubject for reactive state? Are user interactions handled through presentational component outputs? Have I implemented error handling at service and component levels? Are data interfaces defined for component communication? Is takeUntilDestroyed() used for subscription cleanup? Does the data flow follow: Services → Containers → Presentational → User Actions?